Investigando o Cognitive-Driven Development: Percepções sobre Legibilidade e Manutenção de Código

Michel Sales Santana[1], **João Pedro Souza Arruda**[1], **Victor Hugo Santiago Costa Pinto**[1]

[1]Universidade Federal do Pará (UFPA)
66.075-110 – Belém – PA – Brasil

{michel.santana, joao.arruda}@icen.ufpa.br, victor.santiago@ufpa.br

Abstract. *The complexity of source code directly impacts software readability and maintainability. Cognitive-driven development (CDD) is a method that helps developers manage cognitive complexity through Intrinsic Complexity Points (ICPs). This study investigated how CDD influences the perception of code readability and maintainability. A survey was conducted with 26 developers, who comparatively evaluated code snippets. The results indicate that code developed under the influence of CDD was perceived as more modular, organized, and easier to modify. However, factors such as experience and project context also influenced the choices.*

Resumo. *A complexidade do código-fonte impacta diretamente a legibilidade e a manutenção do software. O Cognitive-Driven Development (CDD) é um método que apoia desenvolvedores a controlar a complexidade cognitiva por meio de Pontos de Complexidade Intrínseca (ICPs). Este estudo investigou como o CDD influencia a percepção de legibilidade e manutenção de código. Um survey foi conduzido com 26 desenvolvedores, que avaliaram comparativamente trechos de código. Os resultados indicam que o código desenvolvido sob a influência do CDD foi percebido como mais modular, organizado e fácil de modificar. Entretanto, fatores como experiência e contexto do projeto também têm influenciado as escolhas.*

1. Introdução

A complexidade de código é um dos principais desafios da construção de software, pois impacta diretamente na manutenibilidade, escalabilidade e confiabilidade do software [Thathsarani and Aluthwaththage 2024]. Conforme o trabalho de [Wyrich et al. 2021], os desenvolvedores de software gastam a maior parte do tempo de trabalho compreendendo o código em vez de escrevê-lo. A produtividade de um desenvolvedor está diretamente ligada à sua capacidade de compreender o código com o qual trabalha. Quando a base de código não é bem compreendida, as tarefas de manutenção e evolução se tornam mais demoradas e complexas, o que compromete a eficiência no desenvolvimento de software [Park 2024]. A dificuldade em compreender um código, dentre outros fatores, está associada à carga cognitiva do desenvolvedor, já que ele precisa lidar com múltiplos elementos simultaneamente, o que pode sobrecarregar sua memória de trabalho [Sweller 1988].

Ao longo dos anos, diversas métricas de qualidade foram propostas para auxiliar na compreensão da complexidade cognitiva de um software, como a complexidade

ciclomática de McCabe [McCabe 1976], *fan-in* e *fan-out*, nível de dependência, acoplamento e coesão, entre outras. No entanto, muitas dessas métricas dependem de ferramentas de análise estática e, frequentemente, apresentam limitações quanto à facilidade de compreensão e aplicação prática.

Estudos anteriores indicam que o CDD pode melhorar a qualidade do código, servindo de guia para os desenvolvedores ao oferecer diretrizes que evitam um crescimento desordenado da complexidade do software. Estudos iniciais indicam que o CDD pode melhorar atributos como coesão e legibilidade do código [Pinto and Souza 2022]. Outro estudo aplicou o CDD à refatoração de código, indicando que a aplicação de restrições cognitivas pode guiar o processo de reestruturação e melhorar a legibilidade do código [Pinto et al. 2021].

Embora o CDD já tenha sido explorado em estudos teóricos e em experimentos preliminares, ainda persistem lacunas importantes quanto à sua influência na experiência do desenvolvedor durante o processo de compreensão do código. A forma como o código é estruturado e organizado afeta diretamente sua legibilidade e compreensibilidade — fatores críticos para sua manutenção e evolução [Oliveira et al. 2020]. A interação do desenvolvedor com o código impacta não apenas sua produtividade, mas também sua eficiência cognitiva, o que evidencia a importância de abordagens que reduzam a sobrecarga mental durante tarefas de leitura e modificação. Neste contexto, o objetivo deste estudo é investigar os efeitos do CDD na experiência do desenvolvedor, analisando suas percepções em relação à legibilidade, à carga cognitiva percebida e às preferências durante atividades de manutenção de software.

Diante desse cenário, este estudo busca responder à seguinte questão de pesquisa: ***O CDD é eficaz na redução do esforço cognitivo necessário para compreender unidades de código durante atividades de manutenção?*** Para abordar essa questão, conduzimos um survey online no qual os participantes avaliaram, de forma anônima, duas versões de um mesmo conjunto de trechos de código: uma versão original e outra adaptada segundo as diretrizes do CDD. Essa metodologia para coleta de dados é amplamente reconhecida como uma abordagem eficaz para coletar dados de uma amostra ampla e obter informações sobre comportamentos, percepções e preferências de uma população específica [Punter et al. 2003]. A fim de mitigar possíveis vieses, os participantes não foram informados previamente sobre qual versão havia sido desenvolvida com base nas diretrizes do CDD. Após a análise comparativa, foram coletadas suas percepções relacionadas à legibilidade, ao esforço cognitivo e à facilidade de manutenção do código.

Os resultados dessa pesquisa reforçam a necessidade de considerar o ponto de vista humano do desenvolvimento de software. Ao investigar a percepção dos desenvolvedores sobre a legibilidade e a capacidade de manutenção de código, este trabalho contribui para o entendimento de como práticas e técnicas impactam diretamente a produtividade dos profissionais envolvidos no desenvolvimento de software.

2. Teoria da Carga Cognitiva no Desenvolvimento de Software

A legibilidade de software pode ser encarada sob a perspectiva da Teoria da Carga Cognitiva (ou *Cognitive Load Theory*, CLT) proposta por [Sweller 1988], essa teoria sustenta que a memória de trabalho no processamento de informações complexas é limitada. Quando uma carga cognitiva é superior à capacidade de processamento humano, o apren-

dizado e a aquisição de esquemas mentais ficam prejudicados. Esses esquemas são essenciais para a aprendizagem, pois ajudam a reconhecer padrões e organizar o conhecimento de forma mais eficiente. [Sweller 1988].

Na prática, um código excessivamente complexo obriga os desenvolvedores a dedicar mais tempo identificando métodos, classes e variáveis relevantes, aumentando a carga cognitiva e dificultando a compreensão. Isso afeta a legibilidade e a qualidade do software porque, seguindo a lógica de [Sweller 1988], a necessidade de entender cada detalhe sobrecarrega o desenvolvedor e prejudica as boas práticas e uma estratégia eficaz de refatoração. Como resultado, a produtividade é reduzida, pois menos tempo para implementar soluções de fato, corroborando o que já se observa em pesquisas empíricas acerca do excesso de tempo gasto na compreensão do código [Wyrich et al. 2021].

Desse modo, ao considerar a complexidade sob o ponto de vista da carga cognitiva, é recomendável reduzir a quantidade de elementos que precisam ser simultaneamente analisados. Isso pode ser feito ajustando o código a fim de diminuir a profundidade de aninhamentos e número de dependências, por exemplo, para o desenvolvedor não precise guardar tantas informações na memória de trabalho durante a análise de códigos. Essa premissa fornece uma base teórica sólida para abordagens como o CDD [Souza and Pinto 2020], ao impor limites à complexidade por unidades do software.

2.1. Cognitive-Driven Development

O CDD é uma abordagem que propõe limitar a complexidade durante o desenvolvimento, fundamentada na CLT. A principal premissa do CDD é atribuir ICPs a elementos do código que exigem maior esforço cognitivo para serem compreendidos. Esses pontos são associados a estruturas como if-else, loops, acoplamentos e funções, etc., permitindo mensurar e limitar a complexidade do código de forma objetiva. Além disso, os ICPs podem ser ajustados conforme o contexto do projeto e a experiência da equipe [Souza and Pinto 2020] [Pinto et al. 2021]. Assim, o desenvolvedor é capaz de contabilizar essas estruturas definidas como IPCs e classificar a complexidade do código e, caso as classes e métodos definidos como IPCs não forem mantidos abaixo do um limite previamente definido pela equipe de desenvolvimento, realizar os ajustes necessários para controlar a complexidade do código, de modo a garantir que o desenvolvedor não seja sobrecarregado por aspectos desnecessariamente complexos.

Estudos preliminares mostram que o CDD pode melhorar os atributos de qualidade de software, como coesão e legibilidade, tanto no início do desenvolvimento de software [Pinto and Souza 2022] quanto em cenários de refatoração [Pinto et al. 2021]. Ferramentas de apoio, como o *Cognitive Load Analyzer* [Pereira et al. 2021], automatizam o cálculo dos ICPs e emitem alertas quando o código excede o limite definido, facilitando a adoção contínua dessas diretrizes. Essa combinação de métricas, práticas de desenvolvimento e suporte por ferramentas sugere que o CDD seja capaz de contribuir significativamente para reduzir a sobrecarga cognitiva, promovendo um código mais sustentável e de fácil manutenção.

Embora pesquisas iniciais indiquem que o CDD melhora a manutenção e a legibilidade do código [Pinto and Souza 2022][Pinto et al. 2021], ainda são escassos os estudos empíricos que investigam como essas melhorias são percebidas diretamente pelos desenvolvedores durante suas atividades cotidianas de leitura e modificação de código.

3. Trabalhos Relacionados

No estudo de [Wyrich et al. 2021] foi investigado o impacto das métricas de compreensibilidade de código exibidas na percepção dos desenvolvedores e no desempenho real em tarefas de compreensão, revelando um efeito de ancoragem significativo, no qual as avaliações subjetivas dos desenvolvedores são influenciadas pelo valor da métrica. O trabalho destaca a importância de utilizar métricas validadas no desenvolvimento de software. A pesquisa reforça a necessidade de investigar mais a fundo as consequências da dificuldade percebida do código sobre sua compreensão e sugere que fatores contextuais sejam considerados e controlados.

De acordo com [Palomino et al. 2024], a Experiência do Desenvolvedor vem ganhando destaque como um conceito essencial na engenharia de software, evidenciando o protagonismo dos desenvolvedores em todas as etapas do desenvolvimento. Nessa perspectiva, A experiência dos desenvolvedores é afetada por diversos fatores, incluindo a complexidade do software.

Sob o prisma da sobrecarga cognitiva, [Noda et al. 2023] enfatizam que o aumento de complexidade — seja por configurações de ambiente dispersas, falta de padronização no código ou documentações pouco objetivas — eleva consideravelmente o esforço mental requerido dos desenvolvedores, prejudicando sua produtividade e satisfação.

Em uma pesquisa recente, [Park 2024] sugere que a produtividade não é só influenciada por fatores técnicos, mas também está ligada diretamente a complexidade cognitiva ao interagir com código.

O estudo de [Park 2024] sugere que ferramentas e práticas que reduzam a complexidade do código podem melhorar significativamente a produtividade dos desenvolvedores. Desse modo, abordagens como o CDD podem atuar como mecanismos de mitigação ao estruturar o código de maneira mais compreensível.

4. Metodologia

Este estudo foi conduzido como uma pesquisa exploratória [Prodanov and Freitas 2013] por meio de um survey, com o objetivo de identificar as percepções e desafios enfrentados por desenvolvedores ao lidar com a legibilidade e manutenção do código sob a abordagem do CDD. A escolha do survey como método de investigação é justificada pela necessidade de coletar percepções subjetivas sobre um tema ainda pouco explorado, analisando a experiência do desenvolvedor em relação à legibilidade e manutenção de código.

Estudos anteriores na área de Engenharia de Software indicam que técnicas como a avaliação heurística têm sido amplamente empregadas para investigar a usabilidade de ferramentas de desenvolvimento, bem como sua influência na compreensão do código-fonte [Faulring et al. 2012]. Embora o presente estudo adote uma abordagem distinta, utilizando um survey em vez de uma avaliação heurística formal, a motivação subjacente permanece a mesma: compreender de que forma diferentes paradigmas e abordagens de programação afetam a experiência do desenvolvedor. Com base nessa perspectiva, o estudo foi orientado pela seguinte questão de pesquisa (RQ1):

- RQ1: O CDD é eficaz na redução do esforço cognitivo necessário para compreender unidades de código durante atividades de manutenção?

4.1. Design da Pesquisa

A presente pesquisa adotou o método de survey online, com o objetivo de atingir um número expressivo de participantes para possibilitar um espaço amostral significativo para a pesquisa. Para assegurar o anonimato das respostas, o formulário não coletava nenhum tipo de dado sensível dos participantes, toda coleta de dados realizada era acerca das perguntas da pesquisa, que serão evidenciadas na seção 4.3. Dessa forma, o questionário foi estruturado de modo a caracterizar o perfil dos respondentes e captar suas percepções sobre os códigos apresentados, priorizando a coleta de dados quantitativos. Além disso, incluiu-se uma pergunta aberta com a finalidade de obter dados qualitativos, permitindo a coleta de impressões subjetivas que enriquecem a análise e auxiliam na interpretação contextualizada das demais respostas. Para conseguir participantes para a pesquisa e possibilitar a realização do estudo, o formulário foi amplamente divulgado em canais de comunicação, entretanto, os principais foram WhatsApp e Facebook, para ser mais fácil de chegar até os participantes da pesquisa e promover mais agilidade durante a participação.

4.2. Seleção do Código

A seleção dos trechos de código utilizados no survey foi realizada de forma sistemática para garantir representatividade e adequação ao objetivo do estudo. Os códigos foram extraídos de repositórios públicos no GitHub[1], uma plataforma amplamente utilizada pela comunidade de desenvolvedores. Os trechos de código selecionados eram em JavaScript devido à sua popularidade e familiaridade entre os profissionais da área. Para assegurar a relevância dos códigos selecionados, priorizamos repositórios com maior número de estrelas, um indicador de qualidade e aceitação pela comunidade.

Além disso, foram estabelecidos critérios específicos para a escolha: o código deveria conter entre 100 e 200 linhas, um tamanho suficiente para avaliar legibilidade e complexidade sem que os participantes precisassem gastar muito tempo na análise, para reduzir a evasão, e ser autocontido para não depender de bibliotecas externas ou módulos adicionais e garantir que toda a lógica esteja no próprio código.

4.2.1. Versões dos Códigos Avaliados

Os participantes avaliaram dois trechos de código equivalentes, com a mesma funcionalidade, mas estruturados de formas diferentes. Para evitar viés, o participante não foi informado previamente qual deles é o original e qual segue as diretrizes do CDD:

- Código A (Sem CDD): Implementado sem preocupações explícitas com carga cognitiva, mantendo o design original do código.
- Código B (Com CDD): Estruturado seguindo os princípios do CDD, visando modularidade e menor complexidade intrínseca.

Para garantir que a versão modificada respeitasse os princípios do CDD, a versão reformulada foi criada respeitando o limite de 6 pontos de complexidade conforme o trabalho de [Pereira et al. 2021]. A contagem dos ICPs foi adaptada para refletir as características dessa linguagem. Cada elemento do código contribui para a carga cognitiva

[1] github.com/trekhleb/javascript-algorithms

adiciona 1 ICP, conforme descrito na Tabela 1. A criação da versão reformulada do código seguiu um processo estruturado para garantir que a complexidade cognitiva permanecesse dentro do limite estabelecido.

4.2.2. Identificação dos ICPs no Código Original

O código original foi analisado detalhadamente para identificar os elementos que contribuem para a complexidade cognitiva. Cada ocorrência dos elementos mencionados na Tabela 1 foi contabilizada, e as seções que ultrapassavam o limite de 6 ICPs foram marcadas para refatoração.

Categoria	Elemento Avaliado	ICPs
Estruturas de Controle e Fluxo	Uso de estruturas condicionais (if-else, switch-case, operadores ternários aninhados).	1
	Laços de repetição (for, while, do-while, forEach com lógica complexa).	1
	Blocos de tratamento de erro (try-catch-finally).	1
Acoplamento e Complexidade Funcional	Funções passadas como argumento (map, filter, reduce, setTimeout, setInterval).	1
	Acoplamento contextual (dependência explícita de módulos/classes/objetos globais).	1
	Uso de herança (class extends, modificações diretas no prototype).	1
	Funções assíncronas complexas (async/await aninhado ou Promise.then().catch()).	1

Tabela 1. Atribuição de ICPs para Código em JavaScript

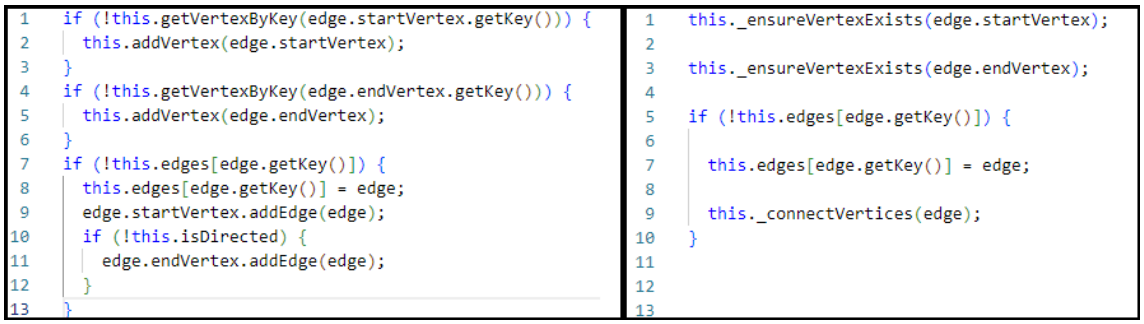
4.2.3. Refatoração para Redução de Complexidade Cognitiva

Trechos de código com alta complexidade cognitiva foram reestruturados utilizando as seguintes técnicas:

- **Modularização:** Funções grandes foram divididas em funções menores e reutilizáveis, reduzindo a carga cognitiva necessária para compreensão.
- **Redução do aninhamento:** Estruturas condicionais profundas (if-else) foram substituídas por *early returns* e *guard clauses*, tornando o fluxo do código mais claro.
- **Simplificação de laços:** Métodos funcionais como map, filter e reduce foram utilizados de forma clara e objetiva, evitando aninhamentos desnecessários.
- **Remoção de acoplamento desnecessário:** A injeção de dependências foi preferida no lugar de importações diretas sempre que possível, reduzindo o acoplamento contextual.
- **Melhoria da organização assíncrona:** O uso de promessas (Promise.then().catch()) e async/await foi reorganizado para minimizar o aninhamento dentro de loops.

4.2.4. Validação das versões refatoradas utilizando os princípios do CDD

Após a refatoração, cada unidade de código foi reavaliada manualmente para garantir que permanecesse dentro do limite de 6 ICPs, conforme a metodologia do CDD. Na Figura 1, o Código A (à esquerda) apresenta uma abordagem sem CDD, com verificações repetitivas e manipulação direta de estruturas de dados, resultando em maior complexidade. Já o Código B (à direita) aplica os princípios do CDD, encapsulando a lógica em funções auxiliares como ensureVertexExists() e connectVertices(), o que melhora a clareza, modularidade e reduz os ICPs. Na tabela 2 é possível verificar com mais clareza a diferença entre os dois códigos. Para realização da pesquisa, foi fornecido ao participante um par de código semelhante ao apresentado, um com a intervenção do CDD e outro sem. Com essa base, o participante era direcionado a responder as perguntas necessárias para a conclusão do questionário.



```
if (!this.getVertexByKey(edge.startVertex.getKey())) {
  this.addVertex(edge.startVertex);
}
if (!this.getVertexByKey(edge.endVertex.getKey())) {
  this.addVertex(edge.endVertex);
}
if (!this.edges[edge.getKey()]) {
  this.edges[edge.getKey()] = edge;
  edge.startVertex.addEdge(edge);
  if (!this.isDirected) {
    edge.endVertex.addEdge(edge);
  }
}
```

```
this._ensureVertexExists(edge.startVertex);

this._ensureVertexExists(edge.endVertex);

if (!this.edges[edge.getKey()]) {

  this.edges[edge.getKey()] = edge;

  this._connectVertices(edge);
}
```

Figura 1. Exemplo de trechos de código: A (versão original) e B (versão correspondente e refatorada sob princípios do CDD).

Tabela 2. Comparação entre os códigos A (original) e B (refatorado segundo princípios do CDD)

Código A (Original)	Código B (Refatorado)
Adição de vértices feita diretamente, repetida	A adição de vértices foi extraída para métodos auxiliares
Adição da aresta feita no método principal	Uso de método auxiliar _connectVertices para adicionar arestas
Lógica concentrada em um único bloco	Divisão em métodos menores e nomeados
Baixa coesão e testes mais difíceis	Maior coesão, testes mais simples
Mistura de várias responsabilidades	Aplica o princípio da responsabilidade única

4.3. Estrutura do Questionário

O survey foi composto por 10 (dez) questões, organizadas em duas seções principais: (i) perfil do participante e (ii) avaliação comparativa dos códigos. A maioria das perguntas foi de múltipla escolha, com uma pergunta aberta ao final para possibilitar que o participante pudesse adicionar seus comentários à pesquisa acerca do survey se achasse necessário. Todas as questões podem ser visualizadas na Tabela 3. Os principais objetivos do questionário era instigar o participante a realizar uma comparação entre os dois

códigos e avaliar qual atendia da melhor forma os critérios perguntados de acordo com a perspectiva dele. As principais análises eram acerca da leitura do código e questões estruturais e se visando uma abordagem com um projeto grande, ou seja, um código com um número expressivo de linhas e diretórios, qual das duas abordagens o participante escolheria para seguir em seu projeto.

Seção	Pergunta
(i) Perfil do Participante	1. Qual a sua faixa etária? 2. Qual o seu tempo de experiência em programação? 3. Qual é o seu nível de escolaridade? 4. Quais linguagens você utiliza com mais frequência?
(ii) Avaliação Comparativa dos Códigos	5. Qual código foi mais fácil de entender? 6. Qual código exigiu menos releituras para ser compreendido? 7. Qual código apresentou melhor organização estrutural? 8. Qual código você conseguiria modificar mais facilmente? 9. Se estivesse em um projeto grande, qual abordagem escolheria? 10. Comentários adicionais (resposta aberta).

Tabela 3. Questões do Survey

4.4. Procedimentos

4.5. Participantes e aplicação da survey

O Survey foi direcionado a desenvolvedores de software, sem restrições de idade, experiência profissional e escolaridade, para coletar diferentes perspectivas sobre os códigos. O recrutamento dos participantes ocorreu através da divulgação do formulário online em grupos de redes sociais relacionados ao desenvolvimento de software, buscando assim a maior diversidade de perfis possível.

O survey foi aplicado por meio da plataforma Google forms, disponível durante 19 dias (de 01/03/2025 a 20/03/2025). A aplicação online garantiu um grande alcance geográfico e anonimato completo nas respostas, permitindo maior liberdade aos participantes.

4.6. Análise de Dados

Os dados coletados foram analisados utilizando técnicas estatísticas descritivas, considerando a amostra de 26 participantes. Foram calculadas frequências absolutas e relativas, além de médias e desvio padrão, para interpretar as respostas quantitativas obtidas nas perguntas fechadas.

Devido ao tamanho reduzido da amostra, a análise focou em tendências gerais em vez de inferências estatísticas mais complexas. As respostas da questão aberta foram analisadas manualmente por meio de categorização temática, identificando padrões recorrentes, percepções sobre a legibilidade e manutenção do código e comentários relevantes que complementaram e enriqueceram a interpretação dos dados quantitativos.

Essa abordagem permitiu uma avaliação exploratória da percepção dos participantes sobre os códigos, oferecendo insights iniciais sobre sua influência na legibilidade e manutenção do código.

5. Resultados

Nesta seção, apresentamos os dados coletados a partir do survey realizado com 26 participantes. Os resultados foram agrupados em três categorias: **perfil dos participantes, comparação entre os códigos e análise qualitativa das percepções**.

Tabela 4. Perfil demográfico dos participantes.

Faixa etária		**Experiência em programação**	
18–24 anos	15	Menos de 3 anos	8
25–34 anos	7	3 a 5 anos	11
35–44 anos	4	Mais de 5 anos	7
Escolaridade		**Linguagens mais utilizadas**	
Ens. Médio (em andamento / completo)	3	JavaScript	20
Graduação (em andamento / completa)	20	Python	15
Pós-graduação (mestrado ou doutorado)	3	Java	7
		PHP	6
		Outras	13

Os participantes avaliaram dois trechos de código em diferentes critérios. A Tabela 5 apresenta os resultados quantitativos da comparação.

Tabela 5. Percepção dos participantes quanto aos critérios de qualidade dos códigos

Critério	**Código A**	**Código B**	**Ambos**	**Nenhum**
Mais fácil de entender	0	15	10	1
Menos releituras necessárias	1	18	6	1
Mais organizado	2	15	9	0
Mais fácil de modificar	4	13	8	1
Melhor para um projeto grande	2	9	15	0

6. Discussão

Os resultados obtidos a partir das 26 respostas do survey evidenciam uma tendência significativa de preferência pelo Código B, desenvolvido com base nos princípios CDD. Em relação à facilidade de compreensão, 57,7% (15 participantes) apontaram o Código B como mais fácil de entender, enquanto 38,5% (10) consideraram ambos igualmente fáceis e apenas 3,8% (1) escolheu o Código A.

No critério de menor necessidade de releitura, o Código B também se destacou: 69,2% dos participantes (18) indicaram que ele exigiu menos releituras, seguido por 23,1% (6) que apontaram equivalência entre os dois e apenas 3,8% (1) preferindo o Código A. Quanto à organização estrutural, 57,7% (15) atribuíram melhor estrutura ao Código B, com 34,6% (9) considerando ambos igualmente organizados e apenas 7,7% (2) escolhendo o Código A.

Em relação à facilidade de modificação, 50% (13) preferiram o Código B, 30,8% (8) indicaram equivalência e 15,4% (4) optaram pelo Código A. Ao considerar qual abordagem escolheriam em um projeto maior, a maioria (57,7%, ou 15 participantes) afirmou que dependeria do contexto. Ainda assim, 34,6% (9) escolheriam o Código B, enquanto apenas 7,7% (2) prefeririam o Código A.

As respostas abertas reforçam essa percepção favorável ao Código B. Vários participantes destacaram que ele "parece mais bem dividido", "facilita a modificação sem retrabalho" e "é mais intuitivo para quem está aprendendo". Um dos participantes afirma: "Quando pego um desses num projeto, é mais fácil analisar só um trecho se sei onde começa e termina facilmente." Outro participante destacou que o uso de funções auxiliares e menor acoplamento contribuiu para uma estrutura mais compreensível.

No entanto, alguns comentários relativizam essa preferência. Um participante apontou que o Código A parecia mais completo, cobrindo mais casos de uso, e outro afirmou que inicialmente escreveria algo semelhante ao A e só depois faria a refatoração para algo mais próximo do B. Esses relatos mostram que, embora o CDD traga benefícios claros, sua aplicação pode depender do momento do desenvolvimento e da complexidade do código.

Também foi possível observar a influência da experiência dos desenvolvedores. Participantes iniciantes tendem a valorizar mais a estrutura modular do Código B, enquanto desenvolvedores com mais tempo de prática reconhecem nele um padrão que favorece modificações futuras e refatorações previsíveis. Isso está alinhado com os fundamentos da Teoria da Carga Cognitiva e com estudos sobre modularização e produtividade.

Por fim, o fato de cerca de 40% dos participantes afirmarem que a escolha entre os dois códigos dependeria do contexto reforça a ideia de que o CDD é especialmente vantajoso em situações com alta complexidade ou projetos de grande escala. Em códigos menores ou mais diretos, os ganhos de clareza podem ser percebidos, mas a simplicidade da tarefa pode reduzir o impacto prático da abordagem.

A investigação sobre a eficácia do CDD na redução do esforço cognitivo durante atividades de manutenção apontou resultados positivos. A análise das respostas revelou que, embora a maioria dos participantes tenha demonstrado preferência pelo Código B, 40% indicaram que essa escolha dependeria do contexto específico da tarefa. Esse dado reforça a ideia de que o impacto do CDD não é absoluto, mas sim condicionado à complexidade e escala do código em questão.

Os achados sugerem que o CDD influencia a percepção da legibilidade e manutenção do código, mas ainda faltam evidências objetivas. Como próximos passos, realizaremos um experimento com eye tracking para observar como os desenvolvedores leem e compreendem códigos com e sem CDD, permitindo uma análise mais detalhada do impacto cognitivo.

7. Limitações do estudo

A amostra de 26 participantes limita a generalização dos resultados para toda a comunidade de programadores. Ainda assim, por se tratar de uma pesquisa exploratória, o estudo possibilita uma visão inicial sobre a percepção dos desenvolvedores em relação ao uso do CDD. Estudos futuros poderão ampliar a amostra de participantes para fortalecer

a validade dos resultados.

A seleção de JavaScript como linguagem-alvo foi motivada por sua popularidade e ampla adoção na indústria. No entanto, a forma como desenvolvedores percebem a legibilidade e a manutenção do código pode variar entre linguagens. Pesquisas futuras podem investigar o impacto do CDD em outras linguagens para determinar se os achados são consistentes.

Os códigos foram escolhidos de repositórios GitHub bem avaliados, considerando número de estrelas e clareza. Isso ajudou a garantir que fossem exemplos relevantes, mas pode ter influenciado os resultados, já que códigos populares tendem a seguir boas práticas. Estudos futuros podem testar o CDD em sistemas legados ou códigos mais complexos para avaliar seu impacto em diferentes contextos.

Futuras pesquisas podem ampliar o escopo da análise e incluir outros métodos, como experimentos controlados, para validar as descobertas.

8. Trabalhos Futuros

Os resultados obtidos nessa pesquisa indicam que o CDD pode influenciar a percepção do desenvolvedor quanto a legibilidade e manutenção do código, principalmente em projetos de grande escala. Porém, as conclusões são baseadas em dados subjetivos fornecidos pelos participantes no survey. Para aprofundar essa análise, propomos a realização de um experimento controlado utilizando ferramentas de eye tracking para avaliar a leitura e compreensão de códigos com e sem CDD.

O uso de eye tracking permitirá coletar dados objetivos sobre como os desenvolvedores interagem visualmente com o código-fonte, fornecendo evidências empíricas sobre os efeitos do CDD na carga cognitiva. Essa abordagem utilizando eye tracking é uma tentativa de realizar uma análise com critérios mais rigorosos para contribuir com a visão dessa pesquisa e compreender melhor a eficácia do CDD avaliando a perspectiva visual também. Com esse método, a ideia é fornecer um cenário em que o participante avalie os códigos e por meio de mapas de calor da captura do eye tracking do participante apontar se existe uma correlação entre o que foi apontado como mais complexo em um código pelo participante junto com seu mapa de calor visual e o que o CDD aponta como partes que oferecem maior complexidade cognitiva.

Esse experimento ajudará a validar se o CDD realmente pode ser considerada como uma ferramenta para auxiliar na análise e na diminuição do esforço cognitivo necessário para compreender o código e pode complementar os resultados obtidos no survey. Tornando a pesquisa acerca da capacidade do CDD mais rica e promissora, uma análise muito importante já que com o avanço da tecnologia e principalmente com a explosão da Inteligencia Artificial no mundo moderno, os códigos tendem a ficar mais complexos.

Referências

Faulring, A., Myers, B. A., Oren, Y., and Rotenberg, K. (2012). A case study of using hci methods to improve tools for programmers. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*, pages 37–39.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Noda, A., Storey, M.-A., Forsgren, N., and Greiler, M. (2023). Devex: What actually drives productivity: The developer-centric approach to measuring and improving productivity. *Queue*, 21(2):35–53.

Oliveira, D., Bruno, R., Madeiral, F., and Castor, F. (2020). Evaluating code readability and legibility: An examination of human-centric studies. *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Palomino, P. T., Fonseca, M., Souza, J., Toda, A. M., Lisboa, R., Cordeiro, T., Pedro, A., and Dermeval, D. (2024). Aprimorando a experiência dos desenvolvedores (devex) para a implementação bem-sucedida de um design system. In *Anais do Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC.

Park, K.-i. (2024). Assessing software developer productivity and emotional state using biometrics. In *Proceedings of the 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

Pereira, J. H. A., de Souza, A. L. O. T., and Pinto, V. H. S. C. (2021). Cognitive load analyzer: A support tool for cognitive-driven development. In *Anais do Simpósio Brasileiro de Engenharia de Software (SBES)*. ACM.

Pinto, V. H. S. C. and Souza, A. L. O. T. (2022). Effects of cognitive-driven development in the early stages of the software development life cycle. In *Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS)*. SCITEPRESS.

Pinto, V. H. S. C., Souza, A. L. O. T., de Oliveira, Y. M. B., and Ribeiro, D. M. (2021). Cognitive-driven development: Preliminary results on software refactorings. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. SCITEPRESS.

Prodanov, C. C. and Freitas, E. C. (2013). *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. Editora Feevale, Novo Hamburgo, RS, Brasil, 2 edition.

Punter, T., Ciolkowski, M., Freimut, B., and John, I. (2003). Conducting on-line surveys in software engineering. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE)*, pages 80–89. IEEE.

Souza, A. L. O. T. and Pinto, V. H. S. C. (2020). Toward a definition of cognitive-driven development. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285.

Thathsarani, H. A. N. N. and Aluthwaththage, J. H. (2024). A comprehensive metric for evaluating object-oriented software complexity. In *2024 3rd International Conference for Advancement in Technology (ICONAT)*. IEEE.

Wyrich, M., Preikschat, A., Graziotin, D., and Wagner, S. (2021). The mind is a powerful place: How showing code comprehensibility metrics influences code understanding. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE/ACM.